IMPLEMENTATION OF DIFFERENT VARIANTS OF TABLE-BASED FREQUENCY SYNTHESIZERS WITH QUADRATURE OUTPUT IN VHDL

Daniel KEKRT[1], *Milos KLIMA*[1], *Radek PODGORNY*[2], *Jan ZAVRTALEK*[2]

[1]Department of Radio Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Technicka 2, 166 27 Prague, Czech Republic
[2]Department of Telecommunication Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Technicka 2, 166 27 Prague, Czech Republic

kekrtd1@fel.cvut.cz, klima@fel.cvut.cz, zavrtjan@fel.cvut.cz, radek@podgorny.cz

Abstract. *This article describes the modelling and implementation of two different variants of direct frequency synthesizer, and evaluation of the performance of the finished design, in terms of memory and speed efficiency. The frequency synthesizer requirement comes from our complex radio transmission system design. The research activity has been focused on finding an optimal balance between simplicity, speed and memory consumption. The modelling was done in MATLAB environment in floating-point and fixed-point arithmetic, and the actual design was implemented and synthesized using the Xilinx ISE suite. The output has been connected to our customized radio front-end built on the Texas Instruments TRF2443 chip. The front-end output signal has been captured and compared with simulation results.*

Keywords

Direct frequency synthesis, FPGA, logic synthesis, memory efficiency, VHDL.

1. Introduction

With improvements in the performance of FPGA (Field-Programmable Gate Array) devices, it becomes important to implement the basic radio system parts in a digital domain. This enhances circuit design simplicity and also offers numerous opportunities for the later modification of the entire system into specific real-world scenarios. Multiple parts of the radio system can be implemented in a single chip, thereby enhancing the overall efficiency in hardware design.

As a part of complex radio transmission system design, we have implemented a direct quadrature frequency synthesizer in VHDL language for the Xilinx Virtex6 FPGA device, i.e. a synthesizer with the fixed table (sine 1st quadrant table and full sine table). The theoretical part of the paper is dedicated to the precise description of the synthesizer transformation into the fixed point arithmetic. The practical part describes the implementation of the designed synthesizer circuit structure in VHDL language.

2. Direct Frequency Synthesizer Model

The frequency synthesizer supports multiple functions in the radio front-end of the digital transmission system. In particular, it reduces the frequency offset on the detection side that arises when the transmitter $f^{(t)}(t)$, or receiver carrier's frequency, $f^{(r)}(t)$, is not absolutely stable and slightly fluctuates in time. The frequency difference, $f_\Delta(t) = f^{(t)}(t)-f^{(r)}(t)$, causes that the received signal, $r(t)$, to be parasitically modulated by a harmonic signal, $e^{j2\pi f_\Delta(t)t}$, of low frequency, even after analog quadrature demodulation. The impact is, that in the case of linear digital modulations it results in the rotation of constellation plane around its center. In order to mitigate this problem, it is necessary to mix the received signal once more with the anti-phase frequency $e^{-j2\pi f_\Delta(t)t}$. When this frequency is low, it is possible to perform the signal processing digitally in FPGA or in DSP directly, in the receiver (after sampling of signal $r(t)$ at the frequency T_P satisfying the sampling theorem).

The functional block in the digital domain that suppresses the undesirable parasitic modulation is called the frequency offset synchronizer; it is composed of three functional parts. The first part, a digital mixer, is simply a pair of two multipliers - one multiplier is for the in-phase signal component $r[n]$, and the second one for the quadrature component. The second block is an offset

estimator that performs the computation of $f_{\Delta}[n]$ from the received signal using prior knowledge of stochastic properties of an implemented modulation. The last block is a frequency synthesizer. The phase increment samples of estimated parasitic frequency are taken on its input from the digital integrator output of the offset estimator. It generates from them the frequency $\widehat{\varphi}_{\Delta}[n] = 2\pi\widehat{f}_{\Delta}[n] \cdot nT_P$ which is then fed into a digital mixer.

Another application area is a frequency multiplexing system, for transmitting several signals. Here, a set of digital synthesizers in the digital domain creates a hierarchy of sub-carriers that are modulated by different transmitted signals. The whole wave packet is then modulated onto a single carrier. There is a complementary receiver block of the same set of synthesizers that demodulate the received signal back to the original components.

The last major area of application is modeling of terrestrial flat or frequency selective Rayleigh channel with fading. The channel fading is simulated using Jake's simulator composed of a set of frequency synthesizers.

The synthesizer output can be described by Eq. (1):

$$y[n] = e^{j\varphi[n]}, \tag{1}$$

Where $\varphi[n]$ is the current content of the phase accumulator that performs the following integration

$$\varphi[n+1] = \varphi[n] + \varphi_{\Delta}[n], \tag{2}$$

$\varphi_{\Delta}[n]$ represents the input phase increments. Let us denote the initial accumulator value as $\varphi_{\Delta}[0]=\varphi_0$. According to the method of calculating the nonlinearity $e^{j(.)}$, the direct synthesizers can either be the sine table or the polynomial ones. In both cases, the entire period of the harmonic function parts of the nonlinearity of $e^{j(.)}$ is expressed using a quarter period of a sine function. Depending on the value of the highest two bit number $\varphi[n]$, which is called as a quadrant indicator, the quarter period may be either mirrored or assigned a negative sign.

Values of the quarter period sampled by specifically selected phase step are stored in the table in the memory synthesizer. This implementation approach contributes mainly to the enhancement of the speed and simplicity of the system design. On the other hand, disadvantages include both higher memory requirements and certain restrictions on generated frequencies resulting from the fineness of the phase step. On the contrary, it is a case of polynomial variant, where the sinus quarter period is replaced by a solid or semi-continuous polynomial approximation. There is a greater delay between samples $y[n]$ and $\varphi[n]$, because the calculation of the polynomial values takes some time, but the frequency accuracy is much bigger.

We begin to describe the implementation of a simpler synthesizer with the sine table. We characterize the synthesizer whose phase accumulator has a bit width, N_b^{φ}. For the output, the following relation holds, $\varphi_q \in \{0,...,M_{\varphi}-1\}$, where $M_{\varphi} = 2^{N_b^{\varphi}}$. The quarter period must have just $M = \frac{M_{\varphi}}{4}$ so that the entire signal comes out a total of M_{φ} phase values. Equation (2) (in floating point operations) can also be expressed in an analogous form:

$$\varphi_q[n+1] = \varphi_q[n] + \varphi_{\Delta q}[n] \bmod 4M, \tag{3}$$

for fixed point operations. The input $\varphi_q \in \{-2M, -2M+1, ..., 2m-1\}$ (in the form of phase increment) then allows to change the current value of the phase $\varphi_q[n]$ according to the sampling theorem at maximum half period forward or backward. The values of the quarter sine table stored in memory of the synthesizer are determined by the Eq. (4)

$$b_q[\ell] = \left\lfloor \frac{1}{2} + (2^{N_b-1}-1)\sin\left(\frac{\pi\ell}{2M}\right) \right\rfloor \text{ iff } 0 < \ell < M-1, \tag{4}$$

where N_b+1 is a number of bits for quantization of the output signal

$$y_q[n] = g(\varphi_q[n]) b_q[f(\varphi_q[n])]. \tag{5}$$

The functions $f(.)$ and $g(.)$ transform a stored quarter period $b_q[l]$ depending on which quadrant the synthesis is currently running at. The function $f(.)$ mirrors a table if necessary, and function $g(.)$ changes its sign.

3. Circuit Structure

The memory synthesizer together with a detailed description of its outputs is shown in Fig. 2. The circuit implementation is shown in Fig. 1. There are two parallel sections in the design. The first section is used to generate the sine; the second one generates the cosine. Both branches share one phase accumulator and a memory of the quarter sinus period, which is accessed once per clock cycle `ComplexEnvelopeClock` (during this period, each of the sections read one sample from a memory). The memory size has been selected as $M = 256$. It corresponds to the minimum phase step $\varphi_{\Delta\min} = 0{,}00613592$ [rad]. Hence, the input of the block has a form of the phase increments signal in two's complement $\varphi_{\Delta q}^{tc}[n]$ with a maximum width of 10 bits (the introduced scalar will be considered as a decimal representation of binary numbers in two's complement notation, i.e. as a whole positive number). It is described in the scheme as `PhaseIncrementIn` and subsequently it is added to the current content of the phase accumulator. Its 10 bits wide output is then split into two parts. Lower 8 bits are the bases for calculating the address of a sinus quarter period memory and upper two bits represent a quadrant pointer. Since the cosine is

shifted to sine by one quadrant $\left(\frac{\pi}{2}\right)$, it is necessary to increment its quadrant pointer in its branch. In contrast, the quadrant pointer is left unchanged in the sine branch. The value of the lowest bit of both quadrant pointers (9th bit) decides in the calculation of address whether or not the 8-bit base $a^{ub}[n]$ will be mirrored (the scalar will be considered as a decimal representation of an unsigned binary number, i.e. as an integer). When the 9th bit has log. '1' the 8-bit base $a^{ub}[n]$ is mirrored. The mirroring process is implemented by using the XOR gate, where the base is negated, and the adder that gets incremented. This will give the address $A^{ub}[n] = M - a^{ub}[n]$. When the 9th bit is log. '0 ' the base is left unchanged and becomes directly to be resulting address $A^{ub}[n] = a^{ub}[n]$. The upper bit of both quadrant pointers (10th) indicates a sign. The top carry bits of adders are generated by mirroring the base $a^{ub}[n]$ and they will be denoted as the saturation bits. When the saturation bit has log. '1', the corresponding part is just at its maximum, i.e., at +1 or -1.

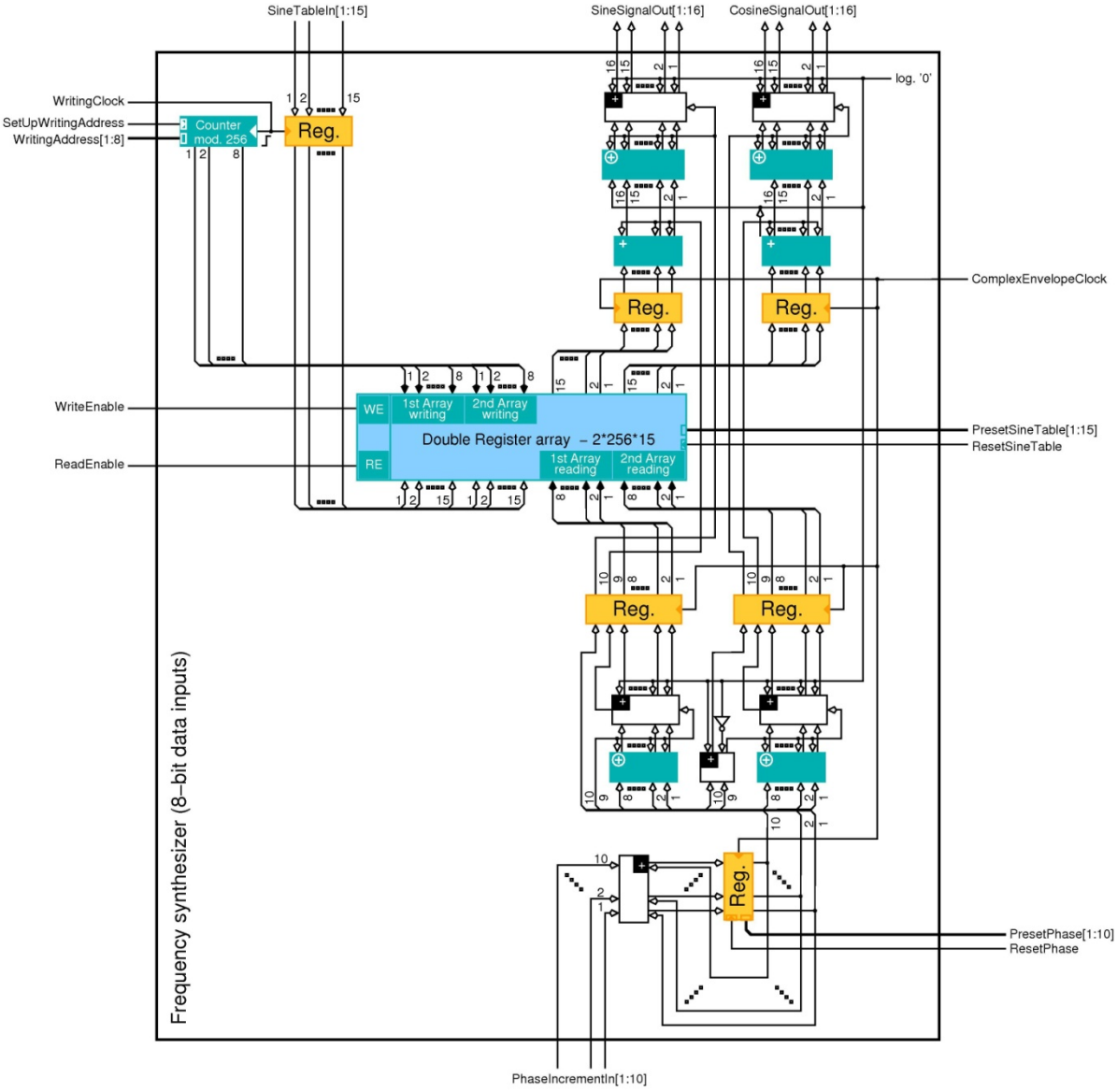


Fig. 1: Synthesizer structure.

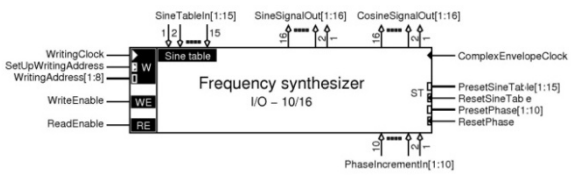


Fig. 2: Synthesizer schematic sign.

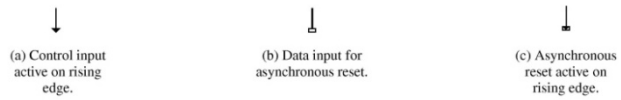
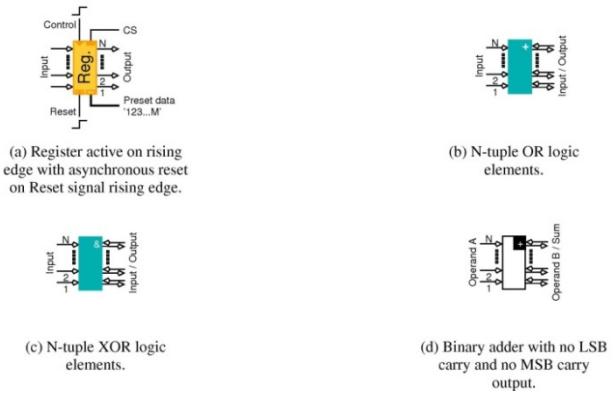


Fig. 3: Signatures of control and data inputs and outputs of circuits used.



(a) Register active on rising edge with asynchronous reset on Reset signal rising edge.

(b) N-tuple OR logic elements.

(c) N-tuple XOR logic elements.

(d) Binary adder with no LSB carry and no MSB carry output.

Fig. 4: Logic and arithmetic circuits used.

The calculated address pairs $A_{\sin}^{ub}[n]$ and $A_{\cos}^{ub}[n]$ together with the sign and saturation bits are buffered by a pair of 10 bits wide addressing registers.

It is clear (from Eq. (4)) that in the memory is stored only the sine table from 0 to $\frac{\pi}{4} - \varphi_{\Delta\min}$ and the sine maximum +1 is missing there. It is therefore necessary to generate it, if necessary. The saturation bits connected to a pair of OR gates behind the output registers are there for that purpose. The rest of the structure performs a sign change of a function if corresponding sign bit is in log. '1'. The total delay of the signal in the synthesizer is two clocks of `ComplexEnvelopeClock`.

4. VHDL Implementation and ISim Simulation

We have implemented the design in VHDL [3], [4] and synthesized the design using the Xilinx ISE 12.3 design suite.

The VHDL entity and signals are shown:

```
entity fsynth_t is
generic (
        sample_w: integer;
        sample_count_w: integer;
        input_w: integer
);
port (
        reset: in std_logic;
        clk: in std_logic;
        input: in std_logic_vector(input_w-1
downto 0);
   sinoutput: out std_logic_vector(sample_w-1
   downto 0) := (others => '0');
   cosoutput: out std_logic_vector(sample_w-1
   downto 0) := (others => '0');
   mem_sin_addr: out
   std_logic_vector(sample_count_w-1 downto 0);
   mem_sin_data: in std_logic_vector(sample_w-1
   downto 0);
   mem_cos_addr: out
   std_logic_vector(sample_count_w-1 downto 0);
   mem_cos_data: in std_logic_vector(sample_w-1
   downto 0)
);
end;
```

We used the VHDL's feature of generic declarations. Therefore, the synthesizers can be easily modified for different bus widths and precision. The entity contains ports for input, quadrature output, clock and reset and buses for interfacing with memory containing sine samples, which is external to the synthesizer.

The two variants of main processes are shown:

```
process(clk, reset, input)
   variable ph: unsigned(sample_count_w-1 downto
   0) := (others => '0');
begin
       if reset='1' then
              sinoutput <= (others => '0');
              cosoutput <= (others => '0');
              ph := (others => '0');
              mem_sin_addr <= (others => '0');
              mem_cos_addr <= (others => '0');

       elsif rising_edge(clk) then
     mem_sin_addr <= std_logic_vector(signed(ph)
     + signed(input));
     mem_cos_addr <= std_logic_vector(signed(ph)
     + signed(input));

              sinoutput <= mem_sin_data;
              cosoutput <= mem_cos_data;

              ph := unsigned(signed(ph) +
signed(input));
       end if;
end process;

process(clk, reset, input)
   variable ph: unsigned(sample_count_w-1 downto
   0) := (others => '0');

   variable sin_addr: unsigned(sample_count_w-1
   downto 0) := (others => '0');

   variable cos_addr: unsigned(sample_count_w-1
   downto 0) := (others => '0');

   variable sin_qp: unsigned(1 downto 0) := (
        others => '0');

   variable cos_qp: unsigned(1 downto 0) :=
   (others => '0');

   variable sin_out: signed(sample_w downto 0)
   := (others => '0');

   variable cos_out: unsigned(sample_w downto 0)
   := (others => '0');

   variable tmp_vect: unsigned(sample_count_w-2-
   1 downto 0) := (others => '0');

   variable tmp_vect2:
   std_logic_vector(sample_w-1 downto 0) :=
   (others => '0');

begin
       if reset='1' then
              sinoutput <= (others => '0');
              cosoutput <= (others => '0');
              ph := (others => '0');
              mem_sin_addr <= (others => '0');
              mem_cos_addr <= (others => '0');

       elsif rising_edge(clk) then
```

```
                ph := unsigned(signed(ph) +
signed(input));

                sin_addr := ph;
      tmp_vect := (others => ph(sample_count_w-
      2));
      sin_addr := "00" &
      (sin_addr(sample_count_w-2-1 downto 0) xor
      tmp_vect);
      sin_addr := sin_addr +
      unsigned(ph(sample_count_w-2 downto
      sample_count_w-2));
      sin_qp := sin_addr(sample_count_w-1 downto
      sample_count_w-1-1);

                cos_addr := ph;
      cos_qp := cos_addr(sample_count_w-1 downto
      sample_count_w-1-1) + "1";
                tmp_vect := (others =>
cos_qp(0));
      cos_addr := "00" &
      (cos_addr(sample_count_w-2-1 downto 0) xor
      tmp_vect);
      cos_addr := cos_qp(1) &
      (cos_addr(sample_count_w-2 downto 0) +
      unsigned(cos_qp(0 downto 0)));
      cos_qp := cos_addr(sample_count_w-1 downto
      sample_count_w-1-1);

                sig_sin_qp <= sin_qp;
                sig_cos_qp <= cos_qp;

      mem_sin_addr <= std_logic_vector("00" &
      sin_addr(sample_count_w-2-1 downto 0));
      mem_cos_addr <= std_logic_vector("00" &
      cos_addr(sample_count_w-2-1 downto 0));

                tmp_vect2 := (others =>
sig_sin_qp(0));
      sin_out := signed("0" & (mem_sin_data or
      tmp_vect2(sample_w-1 downto 0)));
                tmp_vect2 := (others =>
sig_sin_qp(1));
      sin_out := signed(std_logic_vector(sin_out)
      xor tmp_vect2);
      sin_out := sin_out + signed(sig_sin_qp(1
      downto 1));

                tmp_vect2 := (others =>
sig_cos_qp(0));
      cos_out := "0" & (unsigned(mem_cos_data) or
      unsigned(tmp_vect2(sample_w-1 downto 0)));
                tmp_vect2 := (others =>
sig_cos_qp(1));
                cos_out := cos_out xor
unsigned(tmp_vect2);
      cos_out := cos_out + unsigned(sig_cos_qp(1
      downto 1));

                sinoutput <=
std_logic_vector(sin_out);
                cosoutput <=
std_logic_vector(cos_out);
        end if;
end process;
```

Both processes are typical clocked circuits with asynchronous reset. An important part of reset is the setting of memory addresses to zeros for both cases. It is necessary because the synthesizer needs the data from memory ready (the memory is clocked) in the very next cycle after the reset is released.

The simple variant is first and it is quite self-explanatory - the ph variable holds the current phase pointer and is combined with the input to address the memory containing the whole period of sine signal. This solution contains only a minimum logic circuitry needed to perform the correct memory addressing and relaying of output. The main disadvantage is the requirement of a large memory size that is needed to contain the whole sine period. The size of the memory, of course, rises with added precision in both time and amplitude.

The second variant logic scheme is much more complex, but we only need one quarter of the sine which subsequently leads to lower memory requirement. The ph variable is used as in the first case, but it is not used to address the memory directly. Instead of that, the two most significant bits are used to determine which quarter of the sine should be used and whether to invert the result sign. We call it as the quadrant pointer. Corresponding variables have the *_qp suffix*. The temporary vectors are used because of VHDL's inability to perform some types of conversions. The usage of other variables is mainly given by the programming comfort and could be avoided, but the code's readability would drop.

In order to be able to communicate with the device, a set of memories and UART controller are also implemented. This fact, of course, raises the number of used flip-flops. The implementation contains a memory made of FPGA slices for simplicity. The final production-grade system will contain the memory as a block-RAM. We also plan in the final design to slightly change the structure, so it can be implemented as a single-clock synchronous system.

The prepared data set from MATLAB has been downloaded to the synthesized design using a custom Python utility which communicates using our proprietary protocol over USB.

5. The TX Path Circuit Structure

A TX path is a transmitter part of transceiver on Fig. 5 and 6. The input signals are I, Q modulation data and output signal is a RF signal connected either directly to an antenna or to an up-converter.

The path consists of:

- digital I/O interface,
- dual channel DA converter & Clock generator,
- IQ modulator,
- TX LO Synthetiser & Reference frequency generator,
- digital variable attenuator,
- microcontroller.

Fig. 5: Front side of the radio front-end.

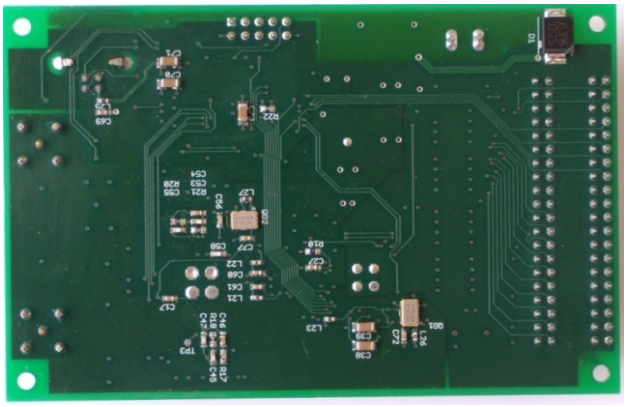


Fig. 6: Back side of the radio front-end.

Some TX path parameters are listed in Tab. 1. The digital I, Q modulation signals are transferred from the FPGA to the digital I/O interface in two's complement form. The functions of the I/O interface include: (1) terminating TX data bus to reduce digital noise, (2) distributing TX data clock and sync signals between DA converter and FPGA and (3) providing connection between the microcontroller and FPGA.

TX data are then transferred to the Dual channel DA converter, which performs the digital 1x/2x/4x interpolation and conversion to analogue I, Q signals. Both DA converter channels have digital gain control, allowing compensation of gain mismatches of I, Q paths.

In order to achieve optimal performance of the DA converter, DA Clock generator with <1ps total clock jitter is used. Therefore, the TX path SNR degradation due to total clock jitter is negligible.

The I, Q signals pass through the 1st order low-pass filter to the IQ modulator. A TX carrier is generated by TX LO synthesiser, which is composed of a high-frequency voltage controlled oscillator (VCO, around 2720 MHz), integer phase locked loop (PLL) and programmable 8/16 output divider which generates TX carrier. It is possible to generate the TX carrier in the frequency range from 165 to 175 MHz, resp. from 330 to 350 MHz.

The IQ modulator drives a variable attenuator followed by the output amplifier.

All TX path blocks are controlled by the microcontroller, which provides following functions: (1) controls the TX path blocks, (2) communicates with FPGA, (3) provides TX path calibration, (4) allows connection to a PC for debugging purposes.

Tab.1: Radio front-end parameters summary.

DA converter (Analog Devices AD9861)		
parameter	value	unit
Resolution	10	bits
Data clock rate	50	MHz
Interpolation factor	1/2/4	-
DAC update rate	50/100/200	MHz
SNR	60.8	dB
SINAD	60.7	dB
SFDR	76	dBc
DA Clock generator (TXC 7C-50.000MBB-T)		
parameter	value	unit
Frequency	50	MHz
Frequency stability (-10 to +70 ºC)	±50	ppm
Aging	max. ±3	ppm/year
Total clock jitter	<1	ps
Transmitter (Texas Instruments TRF2443)		
parameter	value	unit
TX frequency range, div 8	330 to 350	MHz
TX frequency range, div 16	165 to 175	MHz
Output power on RF-TX connector	-25 to +10	dBm
Carrier leakage	-55	dBm
Output noise floor	-139	dBm/Hz
Uncalibrated side-band suppression	-50	dB
TX Reference frequency generator TXC 7C-20.000MBB-T		
parameter	value	unit
Frequency	20	MHz
Frequency stability (-10 to +70 ºC)	±50	ppm
Aging	max. ±3	ppm/year

6. Results

The finished design was synthesized for the Xilinx Virtex 6 FPGA and tested on the real hardware. The final design contains approximately 3000 slice registers, 10000 slice LUTs - approximately 6000 is used as a memory for the memory-intensive case. For the quarter-sine version, the memory consumption is reduced by three fourths. The total number of LUTs used for either case is roughly comparable so the only real difference is the memory requirement. These numbers are skewed because of the peripheral structures we had to implement as well, in order to be able to communicate with the device and read/write memory contents. We have successfully verified the design using the measuring workbench shown in Fig. 5 and we consider it to be fully functional. The operating frequency was 100 MHz, but the synthesizer in Xilinx ISE 12.3 timed the circuit to be able to run up to 320 MHz. The measured waveforms and spectra are shown in Fig. 7 and 8.

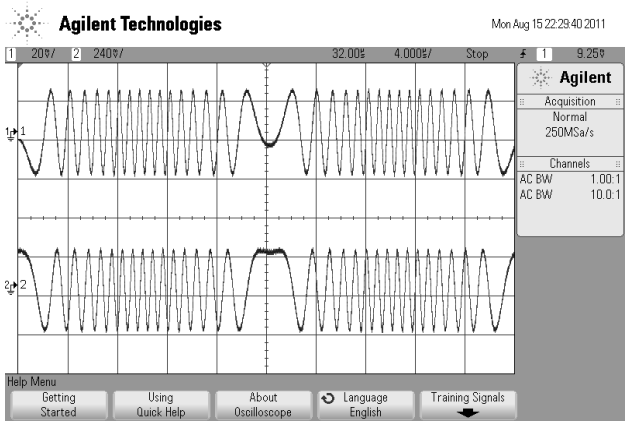


Fig. 7: Measurement – time domain.

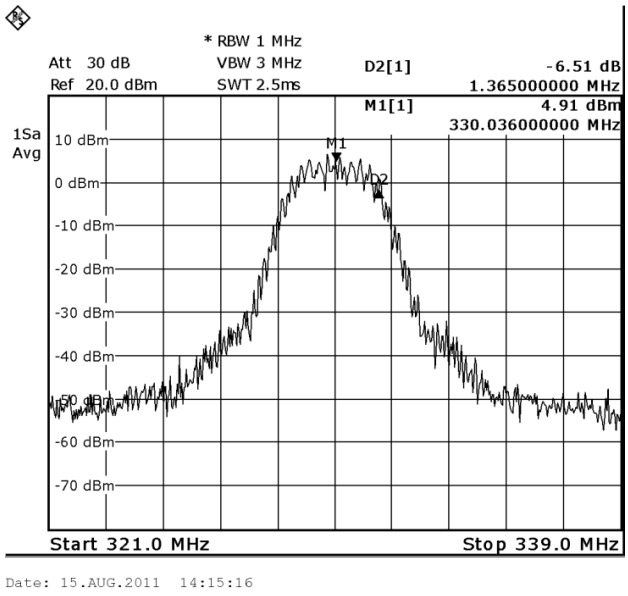


Fig. 8: Measurement – spectral domain.

We have verified the circuit functionality using Tektronix DPO 4032 oscilloscope and Rohde&Schwarz FSL6 spectral analyzer.

7. Conclusion

The key benefit of the described direct quadrature frequency synthesizer with a sine table is the speed (overall structure delay is 2 clocks for both variants) and the frequency stability which is, by the way, a positive feature of all direct synthesizers. However, it requires relatively large amount of memory. The memory should not only be large enough to accommodate a complete quarter period, but also sufficiently quick to be able to serve a double read requests on different addresses in one clock cycle. This is not possible in our Virtex6 FPGA so we opted for doubling the memory. In such a way, we can accommodate the need for two reads in a single clock. It, of course, doubles the amount of needed memory. Another limitation arises from a selected minimum phase step, $\varphi_{\Delta \min}$, that clearly sets a range of the generated frequencies. So when the required memory size and corresponding frequency accuracy is not appropriate or speed is not crucial the polynomial implementation is more convenient. This structure has overall delay 3 clocks in the case of quadrature approximation a 4 clocks with cubic approximation.

Acknowledgements

This research work has been supported by the research project G1 1644 "Doplneni vyuky o moderni principy Turbo kodu a pokrocilych modulaci" of FRVS of Czech Republic and grant No. SGS 10/275/OHK3/3T/13 (Grant Agency of the Czech Technical University in Prague).

References

[1] SKALICKY, Petr. *Cislicove systemy v radiotechnice*. Praha: Vydavatelstvi CVUT, 2003. ISBN 978-80-01-02854-2.

[2] PROAKIS, J. G. *Digital communications*. 5th ed. Boston: McGraw-Hill, 2008. ISBN 978-0072957167.

[3] ASHENDEN, P. J. *The designer's guide to VHDL*. San Francisco: Morgan Kaufmann, 2002. ISBN 1-55860-674-2.

[4] PINKER, J. and M. POUPA. *Cislicove systemy a jazyk VHDL*. Praha: BEN-Technicka literatura, 2006. ISBN 978-80-7300-198-5.

About Authors

Daniel KEKRT, born in Prague in 1981, he received the M.Sc. degree in electrical engineering from the Czech Technical University in Prague (CTU), in 2005. He is now working towards his Ph.D. at CTU. His research interests include image processing, iterative detection, 1D and 2D iterative detection networks and joint iterative synchronization and detection.

Milos KLIMA, born in Prague in 1951, he graduated from the Czech Technical University in Prague (CTU) in 1974. Recently he is a Full Professor at the same university as Head of Department of Radioelectronics, Faculty of Electrical Engineering. He is active in the field of image processing for multimedia and security technology. He is a member of IEEE, SPIE and Czech and Slovak Society for Photonics.

Radek PODGORNY, born in Prague in 1981, he received the M.Sc. degree in electrical engineering from the Czech Technical University in Prague (CTU), in 2005. He is now working towards his Ph.D. at CTU. His research interests include image processing and quality assessment.

Jan ZAVRTALEK, born in Zlin in 1984, he received the M.Sc. degree in electrical engineering from the Czech

Technical University in Prague (CTU), in 2008, and M.Sc. degree in econometrics and operations research from the University of Economics in Prague, in 2010. He is now towards his Ph.D. in CTU. His research interests include wireless sensor networking and power management in radio systems. He is also active in electronic circuit design.